Key Management in a Non-Trusted Distributed Environment

Ed Dawson, Andrew Clark, and Mark Looi

Information Security Research Centre, Queensland University of Technology, GPO Box 2434, Brisbane, 4001, Australia. Email: {e.dawson,a.clark,m.looi}@qut.edu.au

Abstract

Despite the fact that the World Wide Web is an untrusted environment, increasing use is being made of this network (the Internet) in electronic commerce applications. To prevent attacks a strong security architecture is required. A fundamental part of such an architecture is a method for key management. This paper discusses the various components of cryptographic key management especially in relation to the World Wide Web environment. Issues and problems with key generation, key distribution and key storage are raised. An overview is presented of key management systems in several security architectures including SSL, Kerberos and Sesame.

Key words:
Key management, key distribution, key storage, random number generator, World Wide Web.

1 Introduction

Many modern applications are reliant upon cryptographic systems for the provision of security services such as confidentiality, integrity, authentication and non-repudiation. Regardless of whether the type of cryptographic algorithm used is symmetric or asymmetric, and regardless of the specific algorithm used, all cryptographic systems are reliant on the use of one or more cryptographic keys. An important component of such a system is a secure method to manage these keys.

With the increased network connectivity available today, applications such as electronic commerce, home shopping, banking and share trading are becoming more widely used. These applications are typically being provided using a non-trusted distributed network environment such as the Internet. The World Wide Web is perhaps the most common method for delivering such applications, with a wide range of transactional applications being available today.

This article discusses the concepts of secure cryptographic key management in a non-trusted distributed environment such as the World Wide Web. It commences with a brief discussion on the principles of secure key management and why secure key management is a problem in a non-trusted distributed environment. A discussion is given of the three distinct areas of key management, namely key generation, key distribution and key storage, with reference to this particular environment. Examples of how key management is used in selected applications, and examples of current schemes are provided.

2 Principles of Secure Key Management

Key management plays a crucial role in any security architecture. Unless secure key management techniques are used, a security system may be exposed to attack despite the fact that strong cryptographic algorithms are used. This is especially the case in an untrustworthy environment such as the World Wide Web. Key management is concerned with the generation, distribution and storage of cryptographic keys.

Additionally, each of these processes must be controlled by a suitable security policy. It should be noted that prior to designing a key management scheme for a security architecture a security policy for this architecture should be provided. This policy should identify the threats for which the security architecture is aiming to guard against.

In relation to key management this security policy should identify the procedures to be conducted as well as describe the responsibilities and accountability of the different users. In addition the security policy should identify the audit requirements.

2.1 Key Generation

Secure key generation is the foundation of any key management scheme. Keys should be generated in an unpredictable fashion. Cryptographic keys should be generated in such a manner that each of the possible keys is easily likely to occur. The keys should be generated in such a fashion that knowledge of one key should not provide any knowledge about any other keys. The key generation process should involve the use of random or pseudorandom bit sequences. The most important measure of random sequences for cryptographic applications is entropy which gives an indication of the number of independent bits in a sequence, i.e. a sequence of n bits from a truly random source has entropy equal to n bits. A more detailed description of key generation will be presented in Section 4.

2.2 Key Distribution

Cryptographic keys need to be distributed between two users in a secure fashion. If two users already share a cryptographic key then this process is greatly simplified. However, in a distributed environment it may not be reasonable to assume that two users share a secret key initially. Hence the process of key distribution in such a network is more difficult. It may not be possible to use a trusted courier to transfer keys between two users. As well in many cases keys should be distributed in such a fashion that compromise of one key should not influence the security of other keys.

Key distribution in many cases involves the use of intricate cryptographic protocols. There are two types of protocols which are used, namely key transport protocols and key agreement protocols. In key transport protocols one user selects the key and transports it securely to the other user. In a key agreement each user contributes to the generation of keys. Key distribution will be described in more detail in Section 5.

2.3 Key Storage

Keys need to be stored in a secure manner in such a fashion that they will not be revealed to an unauthorised person. In general this is a difficult task in a distributed environment if the keys are stored in an insecure workstation. In order to provide a secure storage method it may be necessary to use a tamper resistant device such as

a smart card. In certain circumstances key storage may need to involve the use of auditing processes such as archival and recovery mechanisms as in some cases the keys may need to be stored for archival measures for a long time. The key recovery mechanisms may be needed in case of equipment failure, or they may also be required for legal reasons. As well, key storage may require mechanisms for destroying cryptographic keys when their use is no longer required. A more detailed description of key storage is presented in Section 6.

3 Problems with Non-Trusted Distributed Environments

Key management is difficult in a non-trusted distributed environment. This is caused by the two key factors, *non-trusted* and *distributed*. Key management is significantly easier in an environment where the communications, users and system administrators are trusted. Such trust does not generally exist on the World Wide Web.

On the Internet (and therefore the World Wide Web) network communications (and thus security critical data) can and does travel through a multitude of often unknown and untrusted systems. There is no guarantee that transmitted data has not been intercepted or modified by an untrusted system. There is also no guarantee that any received data did in fact originate from the system that claims to have sent the data.

In such a non-trusted environment, users are also generally untrusted. Users are typically dealing with faceless entities that may not necessarily have any existence outside the electronic environment. It is difficult to trust an entity that you have never met or visited before, whose only proof of legitimacy is that of a pattern of bits in a computer system. This applies not only to the faceless individual, but also to the faceless large organisation. Not only do large organisations (or electronic merchants) have to trust individual users, but individual users equally need to trust the purported organisations. Such trust is generally not forthcoming on the Internet, except by unknowledgeable or naïve users.

It is also often the case that system administrators cannot be trusted. In almost every system, it is possible for the administrator to view all data files stored on that system.

Because of the distributed nature of the World Wide Web, at least some, if not most, of the communication channels, users and systems will be outside the control of the participating entities.

The lack of ability to trust in the above means that not only must cryptographic mechanisms be used to protect the applications, but appropriate security must also be in place to protect the cryptographic keys.

4 Key Generation

The generation of random keying data for cryptographic applications is a difficult task. Sources of truly random data are not always readily available and therefore there is a heavy reliance on pseudo-random number generators for creating sequences of random data for keying ciphers. A pseudo-random generator requires an initial seed, from which the subsequent random data is derived. The security of the generator, then, relies on the randomness and length of the seed and the properties of the particular generation algorithm.

The process of generating random data can be broken into two phases:

Phase 1. The first phase is the generation of seeding (or keying) data for the random num-

ber generator. This data must be obtained from a reliable random source. Depending on the entropy of the source, the bits may be used directly, or they may be hashed with some cryptographically secure algorithm or compressed to remove redundancy.

Phase 2. If the data obtained from Phase 1 is sufficient for the required application then no further processing is required. If more (perhaps even a continuous supply of) data is required then other techniques which utilise deterministic processes must be used. The random data from Phase 1 may be used to seed a pseudo-random generator in order to provide a continuous (but not infinite) sequence of random material. As well, the seed output from Phase 1 may be input into a hash function. This may be the case for example when in order to be confident of generating a key with n bits of entropy it may be required to collect much more than n bits from the random number generator.

It is important to note that the deterministic processes described above in Phase 2 will not increase the entropy. The only method to obtain entropy is to use a random process. Various sources of random bits have been proposed. Dedicated hardware devices can be reliable but may be impractical due to cost or physical computer limitations. On the other hand, software sources are readily available but are less reliable in providing data which is truly random. The distinction between hardware and software sources of random data is somewhat fuzzy. Oftentimes a hardware device (such as a keyboard, a mouse or a hard disk drive) is monitored by a software application which interprets the signals sent from the device and then outputs the random data. The software may also be used to remove redundancy from the received signal if the source is not truly random. In each of the following examples the entropy of the data from the random source should be tested to determine if removal of redundancy is required. Possible sources of random data are:

- **Radioactive sources:** A radioactive element decays randomly over time, emitting radioactive particles. These particles can be counted and the rate of their emission calculated in order to provide highly random data.
- **Semiconductors:** A noisy diode or resistor may be a very effective source of random data. Such devices are not expensive.
- **Disk drives:** In [4], it was shown that there is a random variation in the rotation speed of a sealed hard disk drive due to air turbulence. Using low level timing measurements on a dedicated disk drive proves to be an excellent source of random data. It is important the drive not be used for any other purpose (for example, storing data) since such activity could adversely affect the randomness of the output - either by unintentional use or by a malicious entity influencing the performance of the disk.
- **Mouse movement:** The location of a mouse pointer within a computer screen and the times between movements have been proposed as a good source of random data. This approach may work well if, as suggested in [9], the user is asked to sign their name (using a mouse or joystick). However, it has been pointed out that care must be taken with particular applications since the number of possible mouse strokes required by the application may be small, making the users responses predictable. This is especially important in network applications (for example, Web Browsers) where an attacker may glean information by monitoring the network [4].
- **Keystroke timings:** A user may be asked to type a random sequence of characters while the time between successive keystrokes is recorded and used to generate random data. The randomness of this data depends on the precision and reliability of the clock used for timing and the users' typing characteristics.
- **Network traffic:** The statistics of the traffic

of a computer network has been proposed as a source of random data. This approach should not be considered secure as others with access to the network can monitor and influence the traffic on the network.

- **Microphones:** Background noise obtained from a microphone may provide a suitable source of random data. Also, the port of an unplugged microphone can provide noise. A Sun SPARCStation has the device /dev/audio which can be used easily to gather such data.
- **System time:** Programmers requiring random number generators for software simulations, etc., frequently use the system time as a seed and a system pseudo-random number generator. Such an approach by itself is inappropriate for cryptographic applications since the time is predictable, as are the pseudo-random number generators provided by common compilers.
- **High frequency clock:** The Intel Pentium range of processors contain a 64 bit counter register (called a Time Stamp Counter) which is incremented upon each internal processor clock cycle. For example, on a 200MHz processor the register will be incremented many times every second. Even though any system timer is deterministic (and therefore predictable), if read at irregular intervals (for example, upon a human keystroke) their values may reveal usable entropy.

None of the random sources listed above provide a simple mechanism for collecting good data. It would be useful if the producers of microprocessors (used in all computers) incorporated a small, inexpensive semiconductor capable of generating random bits. It would then be simple for the developers of operating systems to provide a mechanism by which a programmer could easily access the random data.

The difficulty of finding suitable seeding material for random number generators is highlighted by the number of attacks on existing systems which have been published. Typically, the bits obtained from the sources outlined above are not independent. Calculating the entropy of the collected data gives an indication of the number of independent bits. If the entropy is low then the amount of usable random data obtained may be insufficient for a particular application, in which case the random data can be used to seed a pseudo-random number generator which will produce the required amount of information.

Many pseudo-random number generators exist but not all are suitable for cryptographic applications. The algorithm used must have good properties and the seeding data used by the algorithm must be truly random. It is essential that the source of the seed be totally unpredictable to prevent an attacker from guessing it. It is also important that the length of the seed required by the pseudo-random number generator be large enough to prevent an attacker exhaustively trying all seeds. When a cipher is used to generate random data the seed is usually used as a key for the cipher in which case the required length of the seed is dictated by the key size of the cipher. Ciphers are usually designed with keys large enough to prevent them being determined by an exhaustive attack.

Hardware devices such as smart cards may have some random seed data programmed into them at production time or they may have their own genuine noise source on the card (such as a semiconductor of some type). Such data may be used to seed a pseudo-random number generator which runs on the hardware in the card.

It is important to ensure that random numbers generated satisfy randomness properties. Black-box methods to test the generating device for randomness properties are described in [5] and [6]. It is recommended that these procedures be applied as a first-step in analysing the strength of the key generating algorithm. If such an algorithm fails to satisfy these tests, then

it is recommended that it should not be used for the generation of keys where the secrecy of information is paramount. To gain a true understanding of the actual entropy it is necessary to be provided with the actual source code, or complete description of the method, used to generate random numbers. This allows an analyst to conduct an in-depth examination of the technique used to generate random numbers.

5 Key Distribution

In this section we discuss techniques for establishing a secret key between two parties who wish to communicate securely. This secret key is typically used to secure a cryptographic session, i.e. a session key. Typically the key distribution phase incorporates some kind of authentication mechanisms so that each party can be assured of the identity of the other. In many cases key distribution schemes can be generalised to allow for the establishment of a key between two *or more* parties but in this paper we limit ourselves to the case of two parties. As mentioned in Section 2.2, there are two types of key distribution schemes: key transport protocols and key agreement protocols. Each of these schemes are now described in some detail.

5.1 Key Transport Protocols

In key transport protocols one party is responsible for generating the session key and transmitting it securely to the other parties that require it. The key should be generated using a technique similar to the ones described above in Section 4. Secure transmission of the key to the other party can be achieved using either symmetric cryptography or asymmetric cryptography.

When symmetric cryptography is used, a trusted server is required, which shares long-term secret keys with each of the parties wishing to communicate in the session. The trusted server usually generates the required session key and distributes it to each party, protected with the long-term key that is shared with that party. This process is outlined in Example 1. Note that in all the examples below, as is standard when describing cryptographic protocols, A stands for Alice and B stands for Bob. Hence, keys belonging to Alice (A) are referred to as "hers" and keys belonging to Bob (B) are referred to as "his".

Example 1 *Assume parties A and B wish to establish a session key. A shares a long-term secret key, K_{AS}, with the trusted server, S, and B shares a long-term secret key, K_{BS}, with S.*

(1) A sends a message to S requesting a session key for communication with B.
(2) S generates a session key K_{AB}.
(3) S encrypts K_{AB} using K_{AS} and sends the result to A.
(4) S encrypts K_{AB} using K_{BS} and sends the result to B.
(5) A and B decrypt K_{AB} using their respective long-term keys (shared with S).

When asymmetric cryptography is used, no trusted server is required, i.e. the session key can be generated and distributed directly to the other party. The transmitted session key is usually protected using the public key of its recipient. The sender of the key may optionally sign the transmitted data with their private key, providing the recipient with the ability to verify who sent it. An example of a key transport protocol which utilises asymmetric cryptography is the Needham-Schroeder public key protocol [14] which is described now in Example 2.

Example 2 *In this example each party sends a symmetric key to the other. This need not be the case – that is, only one key need be communi-*

cated if that is all that is required. This protocol provides mutual authentication.

(1) A generates a session key K_{AB}.
(2) A encrypts K_{AB} using the public key of B and sends it, along with her identity, to B.
(3) B generates a session key K_{BA}.
(4) B decrypts K_{AB} using his private key and sends both K_{AB} and K_{BA} to A, encrypted under A's public key.
(5) A decrypts K_{AB} and K_{BA} using her private key and verifies that the received K_{AB} is the same as the one sent to B in Step 2.
(6) A encrypts K_{BA} with B's public key and sends it back to B.
(7) B decrypts K_{BA} with his private key and verifies that it is the same as the one sent to A in Step 4.

There are advantages and disadvantages of each of these two forms of key transport in an untrusted distributed environment. One advantage of the first protocol is that the generation of session keys is greatly simplified since it is only necessary to have the trusted server generate these keys. Hence, it is only necessary to place a dedicated device to generate keys as described in Section 4 at this server. However, fundamental problems with this protocol are that a trusted server is required and that each of the users must share a long term secret key with the trusted server. This requirement is not possible in the World Wide Web for many applications especially if one considers the case of international electronic commerce. On the other hand, the second protocol does not have this problem since no trusted server is required. However, there are several disadvantages with this protocol since it involves the use of public key algorithms. These algorithms usually require greater bandwidth and computational effort than symmetric algorithms.

Another problem with public key algorithms is that a Public Key Infrastructure (PKI) is required – for example a certification authority (CA) is needed to certify the public key of each user. A certificate is a signed data structure which provides assurance between an entity and its alleged public key. There are many issues to finalise before a PKI is used on the Internet. For example, the type, format and legal status of certificates need to be resolved. Currently, there are several working groups investigating the design of such certificates including the Internet Engineering Task Force (IETF) [11] and the Open Group [15].

5.2 Key Agreement Protocols

In contrast to key transport protocols, key agreement protocols require input from each of the parties in the generation of the session key. These protocols generally utilise public key cryptography. Probably the best known key agreement protocol is the Diffie-Hellman key agreement scheme [8]. The following example presents a key agreement scheme based on the Diffie-Hellman scheme known as the Station-to-Station protocol (STS) [13].

Example 3 *This example presents the Station-to-Station protocol which gives key agreement and at the same time, mutual entity authentication. Prior to establishing a session key the following public parameters must be set: a prime, p, and a generator, α of $\mathbb{Z}_p^*$ ($2 \leq \alpha \leq p-2$). It is also assumed that A and B each possess a public/private key pair and that each trusts the public key of the other. The protocol then progresses as follows:*

(1) A chooses a random secret x ($1 \leq x \leq p-2$) and sends $\alpha^x \bmod p$ to B.
(2) B chooses a random secret y ($1 \leq y \leq p-2$) and computes $\alpha^y \bmod p$. B computes the session key, K, by raising $\alpha^x \bmod p$ (received from A) to the power y (mod p). B

also forms a signature (using his private key) of α^y and α^x concatenated which he encrypts with the session key, K. B then forwards $\alpha^y \bmod p$ and the encrypted signature to A.

(3) A computes the session key, K, by raising $\alpha^y \bmod p$ (received from B) to the power x (mod p) and then decrypts the encrypted signature (also received from B) which can then be verified. When B's message has been successfully verified, A generates a similar message by forming a signature of α^x concatenated with α^y and encrypting it with the session key, K. A then sends the encrypted signature to B.

(4) B then decrypts the incoming message using K and verifies the signature contained within. If the signature is successfully verified then both A and B can be satisfied that the other has the session key, K.

The protocol in Example 3 has the same advantages as described above for Example 2 in that no trusted server is required. Also, the disadvantages are similar to Example 2 in that the protocol involves public key ciphers so a PKI is required, and the protocol is costly in relation to band-width and computation.

Key agreement seems to be the ideal protocol to use in a distributed environment where no trusted server is available and the two parties who wish to set up a secure communication channel do not trust one another. The reason is that each party is able to have input into the format of the session key. One major problem with key agreement is that each party needs to have a method to generate random numbers. As described in Section 4, this is not a simple process. There are many applications such as home banking where a person is communicating with a bank (which is usually assumed to be trusted) where a key transport protocol may be much more appropriate.

6 Key Storage

Because of the untrusted and distributed nature of the World Wide Web, *key storage* is a difficult problem, but one that must be solved in order to provide a secure key management system, and thus a secure application. As previously discussed, the security of any application, whether distributed or not, hinges upon the protection of the cryptographic keys. It is pointless generating and distributing the keys in a secure manner, if the keys are then stored by the end user in an insecure manner. Two aspects of key storage are described here, namely secure storage devices and secure storage mechanisms.

6.1 Secure Key Storage Devices

The primary threats to the keys caused by inadequate security in the key storage area are that the user's keys may be revealed, obtained and/or misused by an illegitimate person. This could be an intruder who has managed to gain access to the user's key storage area or key storage device, or a system administrator who has access to the files stored on the system, or to programs stored on the system.

It is certainly inadequate to store the keys in the clear (unencrypted) on a device that is accessible or usable by anyone other than the legitimate user. For example, storing the keys on a shared, network hard drive means that the system administrator will most probably have access to the keys. If the hard drive is accessed via the network, any person monitoring the network will potentially be able to intercept the keys as they are being transmitted from the hard drive, over the network, onto the user's workstation.

Even if the keys are stored in encrypted form, it is clear that the keys must be temporarily de-

crypted in order to be used. If this decrypted key were to be made available, albeit temporarily, to an insecure or untrusted workstation, this key could be obtained by an attacker or by a system administrator with appropriate access to the workstation. For a particular application, it must be decided if there is a need to protect the key at all times, ensuring that it is never revealed to an untrusted source.

A common method of protecting keys is by the use of a user specific secure computing and storage device, such as a smart card or a processor based token. In this case, the cryptographic key to be protected is stored on the smart card, and the security of the smart card is established in such a way so that no one can read or retrieve the key from the smart card. Whenever the key is needed (to perform a cryptographic operation), the data to be encrypted or decrypted is sent to the smart card and the cryptographic work is performed by the smart card.

This technique allows the user to securely carry their cryptographic key with them from one location to another, and to store this key in a manner that is reasonably secure. Placing the execution of the cryptographic algorithms on the smart card also means that the cryptographic key is never revealed outside the smart card. As long as trust can be maintained in the smart card, then the key storage problem can be addressed using this technique.

Of course, there are several disadvantages to this technique. A smart card reader is required to interface between the user's smart card and the computing workstation. Every user must be issued with a smart card containing appropriate cryptographic keys (key distribution then becomes an issue). Users are required to possess the smart card (which, depending on the application, may actually be an advantage). The smart cards must thus be managed in a secure manner, imposing additional administrative overheads.

Despite these problems, smart cards seem to offer a practical method to store cryptographic keys in an insecure environment such as the Internet. The cost of smart cards has greatly decreased over the past few years. It is anticipated that in the future, most computers will be equipped with smart card readers, and most operating systems will have functions to access smart cards and smart card readers.

In any system that incorporates the use of smart cards, the security policy must be clear in the responsibilities of each entity as to the use and management of each smart card. Ultimately, the user must bear some responsibility, in this case to properly manage their own smart card.

6.2 Secure Key Storage Mechanisms

In addition to selecting a particular device for key storage, appropriate mechanisms are needed to support the storage of these keys. The application of cryptography to certain communications systems may require complex key storage mechanisms. For example, key recovery mechanisms may be required for legal wire-tapping or for audit trails in secure electronic commerce.

In the case where public key ciphers are used in the security architecture a PKI will be required. As described in Section 5.1 a PKI will need mechanisms to support certificates. A certificate authority structure may greatly increase the complexity and cost of the security architecture. Currently it may cost several hundred dollars to obtain a certificate from a private issuer in the USA [18].

7 Security Architectures

There are a multitude of applications that run in a non-trusted distributed environment, such as the World Wide Web, that require secure key management. In fact, any application that relies on being able to securely identify an entity, or being able to ensure data confidentiality, integrity or authenticity requires secure key management.

Examples of applications that operate in a non-trusted distributed environment that require secure key management include the following.

- Internet banking
- Home shopping
- Credit card purchasing over the Internet
- Share market transactions
- Electronic Mail

A number of security architectures have been applied to each of the examples above. We now give a brief description of some of these architectures highlighting the key management techniques used in each system.

7.1 Secure Sockets Layer

The Secure Sockets Layer (SSL) [7] is a protocol which was originally proposed by Netscape to provide security services for the World Wide Web. SSL is now supported by most WWW browsers and servers and has also been incorporated into a number of other networking applications - for example, TELNET and FTP. SSL provides for mutual authentication using certificates, although in most applications only the server is authenticated. SSL provides no solutions to the key generation problem, instead the application is responsible for generating random data. In fact, an early version of the Netscape Navigator used a predictable seed for its random number generator, leading to a much-publicised attack [10]. The SSL specification suggests two alternatives for key distribution - RSA and Diffie-Hellman. As discussed above, the use of RSA requires one party to generate a session key, where-as Diffie-Hellman requires input from both parties. SSL does not specify any key storage requirements, leaving that task to the implementor of the application.

7.2 Secure Shell

The Secure Shell (SSH) [17,19] is an application used mostly as a secure replacement to TELNET and RSH (remote shell) for connecting to Unix computers over the network. SSH can also be used to securely transfer files between Unix computers and also between client hosts and Unix servers. SSH prevents passwords from being exposed on the network in the clear. As well as allowing the traditional username/password authentication on Unix hosts, SSH also allows users to identify themselves using public key cryptography. SSH provides a tool for users to generate their own public/private key pairs. Depending on the platform of the SSH client (it could, for example, be a Microsoft Windows client or a Unix client), different techniques are used for seeding the random number generator. The F-Secure SSH client for MS Windows initialises its seed using mouse movement. The Unix client gathers environmental information from the operating system which it combines to form a seed. In both cases the seed is stored in a file locally and updated each time keys are generated. In the SSH protocol, the client generates the session key and sends it to the server encrypted under the server's public key. User's RSA private keys are stored on the client computer's file system encrypted using a password provided by the user.

7.3 Kerberos

Kerberos [12] is a security architecture developed by Massachusetts Institute of Technology (MIT) in the mid 1980s. Kerberos is a purely symmetric key based architecture originally developed for a university environment with the aim of providing tighter authentication mechanisms than the traditional password based systems as well as privacy services. Kerberos allows *single sign-on* to the network, relinquishing the user of the need to repeatedly enter username/password information for each new application.

Kerberos makes use of an on-line *Authentication Server* (AS) and *Ticket Granting Server* (TGS) in its architecture. The AS authenticates users and provides them with a *Ticket Granting Ticket* (TGT). This is the sign-on phase in the Kerberos protocol. The TGT contains information about the user and their privileges. When the user wishes to contact a server (using client software), they pass their TGT to the TGS which authenticates the user's request before generating a Server ticket which it returns to the user. The ticket contains a session key which can be used for communicating with the server application. Thus, in the Kerberos environment the TGS is required to generate session keys. Session keys are also generated by the AS (during authentication) and may also be generated by the application server if necessary. In any case, Kerberos never requires the user to generate a session key so only the servers (AS, TGS, application servers) require good key generation facilities.

As with all purely symmetric key based systems, there is the requirement for an initial shared secret. In the case of Kerberos the user shares a password with the AS. This password is used as a symmetric key when the user first contacts the AS. It is also used by the AS in its response to the user.

Kerberos does not define any specific techniques for key generation. Key storage is performed by the AS which must store each of the users passwords for the purpose of authenticating the users and encrypting communications sent from the AS to the user. If the AS is compromised then it is possible to masquerade as any user. The TGS also stores keys in its ticket cache. If the TGS is compromised then all future session keys and possibly past session keys (stored in the ticket cache) can be obtained giving the perpetrator the ability to listen in to, and modify any future sessions between users and servers and to view past sessions.

7.4 Sesame

Similar to Kerberos, Sesame [1] is a distributed security architecture which improves on Kerberos by adding (among other things) public key cryptography. Sesame was developed in a joint research project between Bull, ICL and Siemens – all European based companies. Sesame supports all of the symmetric key protocols that are implemented but has also been extended to incorporate the use of certificates.

Sesame uses the same "single sign-on" concept as Kerberos through the use of the Authentication Server (AS) which is part of Sesame's Domain Security Server (DSS). In the Sesame architecture, the DSS returns to the user a Privilege Attribute Certificate (PAC) which defines the users privileges. The PAC must be submitted to the application server on connecting. An additional feature of Sesame is that it makes use of role-based access control (RBAC) to restrict users access based on their role in an organisation.

In Sesame, since the user has a public private

key pair, it is possible for the user and the AS to communicate without the need for a prior shared secret (provided that the user trusts the public key of the AS and vice versa). Also, the client can now generate the session key and send it to the application server using their public key. Thus, key generation is now the responsibility of the client, or it can be a shared responsibility with the server using a Diffie-Hellman-like protocol. In any case, the need for a ticket granting server is no longer present. For Sesame, key storage is required by each user and each server in order to store their private key. Sesame optionally supports the use of secure cryptographic tokens (such as smart cards) for this purpose.

Sesame applications are now available for common Internet protocols such as TELNET, RLOGIN and Remote Procedure Call (RPC) [2]. This appears to provide a more secure architecture including better key management than earlier systems such as Kerberos, especially in an Internet environment.

7.5 PGP

PGP [16] (Pretty Good Privacy) is perhaps the most widely used application for securing electronic mail. PGP utilises its own public key mechanism. Rather than using a hierarchical model, PGP uses a peer-based model where users keys are signed by their peers, creating a "web of trust". In this model, for example, if A trusts B, and B trusts C, then A should trust C. There is no one method for distributing the master public keys of each user. Users are responsible for the distribution process of master public keys. This is usually done through a different channel, by electronic mail, through a web page, or physically transferred on electronic media such as floppy disks. These master public keys are used to encrypt message keys, and these message keys are then used to encrypt each individual message.

The Unix version of PGP utilises key-stroke timings (the difference in time between successive key-strokes) to generate a seed before generating a public/private key pair. A random seed file is stored for each user - initially based on the key-stroke timings - which is updated each time a symmetric or asymmetric key is generated. PGP secret keys are stored on the host file system encrypted under a pass-phrase designated by the user.

8 Conclusions

In this paper we have outlined the most important aspects of key management and its requirements for use in Internet applications such as the World Wide Web. The key areas of key generation, key distribution and key storage have each been described in detail and a number of examples of applications using these technologies have been given.

The lack of a global PKI has impeded the widespread use of cryptographic protocols on the Internet. A number of solutions are currently in use but these systems are not gaining wide acceptance due to the lack of a global PKI. A number of hardware devices are available to assist with secure key generation and key storage but the wide-spread use of such devices will take time.

As technology improves, and the problems associated with the development of a global PKI are gradually solved, a greater amount of trust in the use of cryptography to secure applications such as those outlined in this paper will be achieved.

References

[1] P. Ashley and M. Vandenwauver, Practical Internet Security: Overview of the State of the Art and Available Technologies, *Kluwer* (January 1999).

[2] P. Ashley. M. Vandenwauver and B. Broom, A Uniform Approach to Securing Unix Applications Using SESAME, *Australasian Conference on Information Security and Privacy (ACISP'98), in Lecture Notes in Computer Science* **1438** (Springer, Berlin, 1998) 24–35.

[3] William Caelli, Dennis Longley and Michael Shain, Information Security Handbook, (Stockton Press, New York, 1991).

[4] D. Davis, Internet Newsgroup Posting to sci.crypt (Re: Need Advice: RNG) (October 1996).

[5] E. Dawson and H. Gustafson, Statistical Methods for Testing the Strength of Key Generators, *Proceedings of PRAGOCRYPT'96* (Prague, October 1996) 452–466.

[6] E. Dawson and H. M. Gustafson, A Method for Measuring Entropy of Symmetric Key Generators, *Computers and Security* **17(2)** (1998) 177–184.

[7] T. Dierks and C. Allen, The TLS Protocol Version 3.0, *Transport Layer Security Working Group, RFC 2246* (January 1999).

[8] E. Diffie and M. E. Hellman, New Directions in Cryptography, *IEEE Transactions on Information Theory* **22** (1976) 644–654.

[9] Carl Ellison and Burt Kaliski, Cryptographic random numbers Draft v1.0, *http://www.clark.net/pub/cme/P1363/ranno.html, Appendix E of P1363* (November 1995).

[10] I. Goldberg and D. Wagner, Randomness and the Netscape Browser: How secure is the World Wide Web?, *Dr Dobb's Journal* (January 1996).

[11] IETF, The Internet Engineering Task Force, *http://www.ietf.org/*.

[12] J. Kohl and C. Neuman, The Kerberos Network Authentication Service V5, *RFC 1510* (September 1993).

[13] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, Handbook of Applied Cryptography, (CRC Press LLS, Florida, 1997) 489–586.

[14] R. Needham and M. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Communications of the ACM* **21** (1978) 993–999.

[15] Open Group (The), *http://www.opengroup.org/*.

[16] PGP, The International PGP Home Page, *http://www.pgpi.com/*.

[17] SSH, IETF Secure Shell Working Group, Secure Shell (secsh) Charter, *http://www.ietf.org/html.charters/secsh-charter.html*.

[18] Verisign, Software Developer IDs for Authenticode, *http://digitalid.verisign.com/*.

[19] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne and S. Lehtinen, SSH Protocol Architecture, *Internet Draft* (August 1998).